A Bit about Me: Rendering Systems

A Bit about Me: Rendering Systems

Hardware

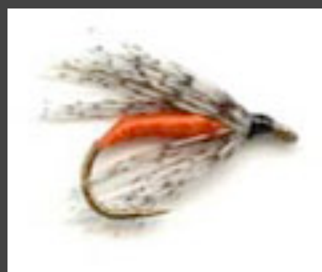
- Pixar Image Computer & CHAP
- REYES Machine & FLAP

Software

- RenderMan
- Real-Time Shading Language
- Spark

A Bit about Me: & Beyond

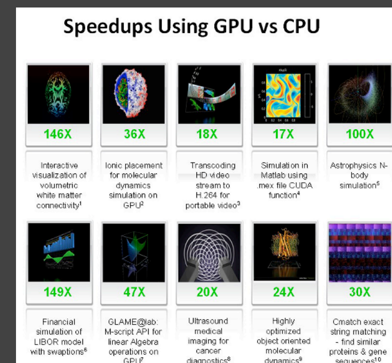
Brook: Stream computing on graphics processors



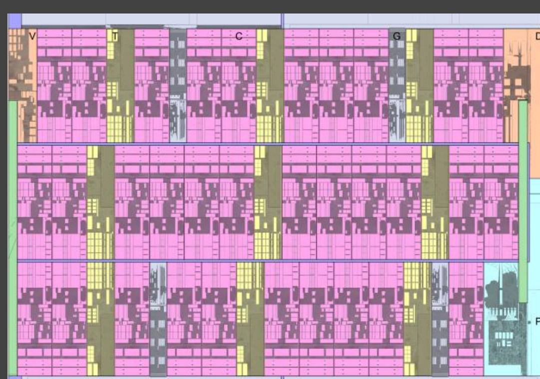
NVIDIA CUDA

OpenCL

GPGPU



Larrabee: An x86 architecture for visual computing



A Graphics Perspective on Co-Design

Pat Hanrahan

DOE Stanford PSAAP Center

Stanford Pervasive Parallelism Laboratory

(Supported by Sun/Oracle, AMD, NVIDIA, Intel, NEC)

Salishan Conference on High Speed Computing

April 25, 2011

Application

The Road to Point Reyes
Lucasfilm

R.E.Y.E.S. = Renders Everything You Ever Saw

Simulating the everyday world

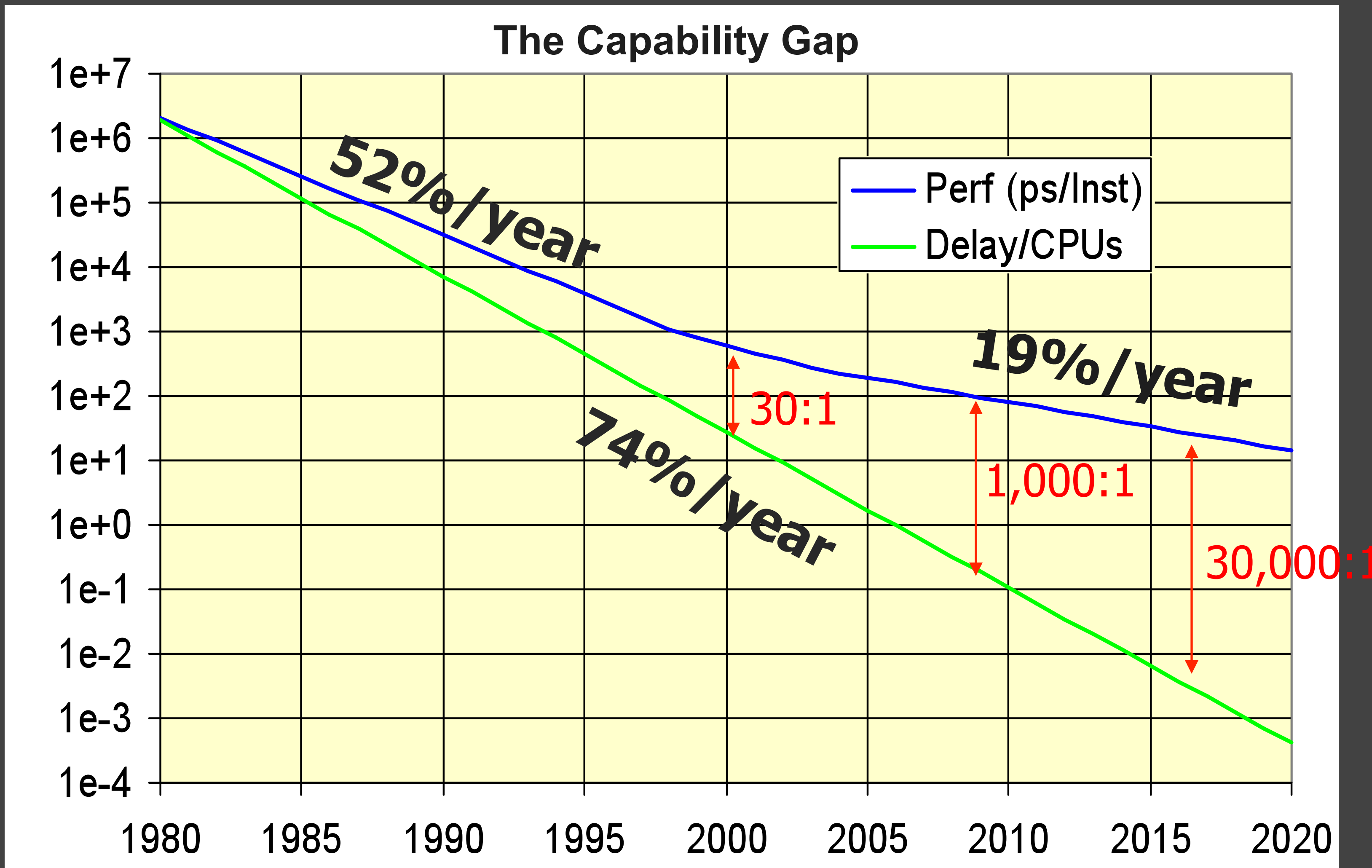
Hardware

REYES Machine Goals (1986)

Pixels	3000 x 1667 (5 MP)
Depth complexity	4
Pixel area of a micropolygon	0.25
Number of micropolygons	80,000,000
FLOPs per micropolygon (minimum)	300
Total calculation	24 GFs
24 frames per second	.576 TFs

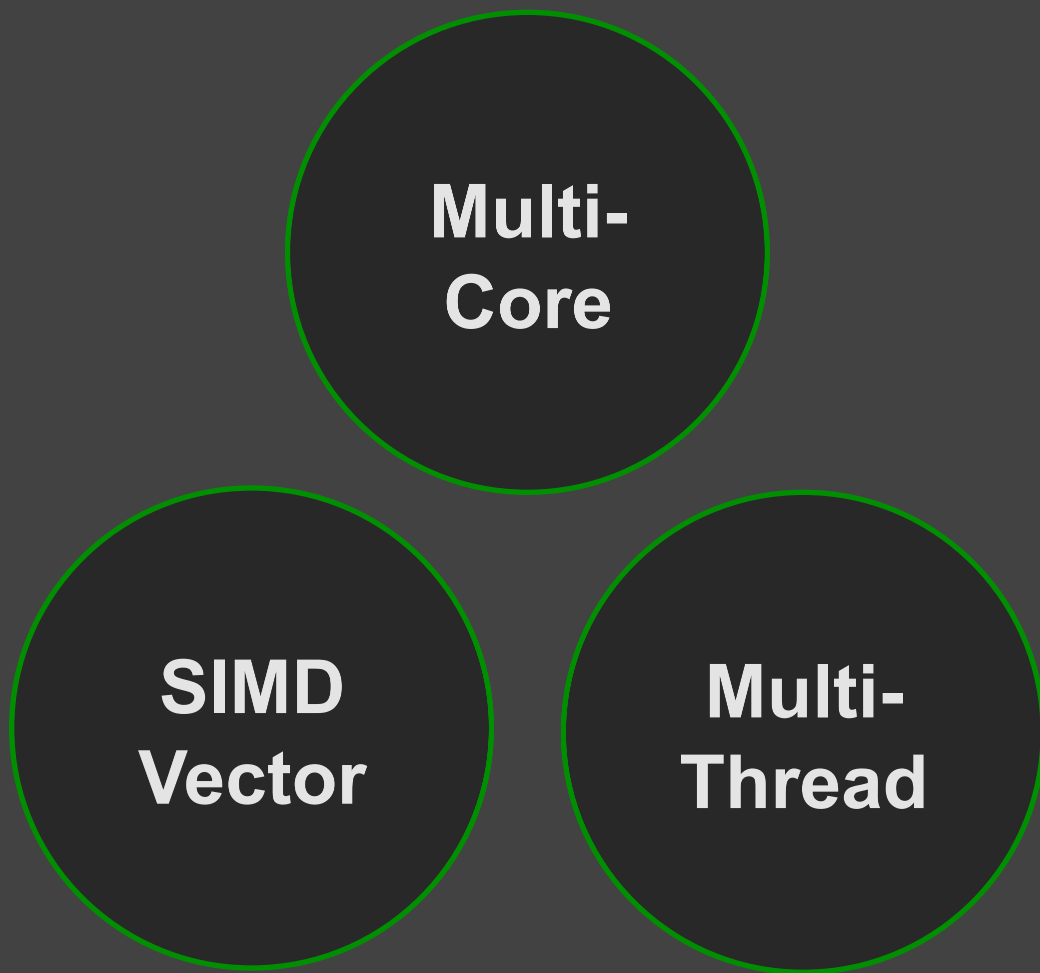
Goal ~ 1 frame in 2 minutes – real-time was inconceivable

CPUs Waste Resources

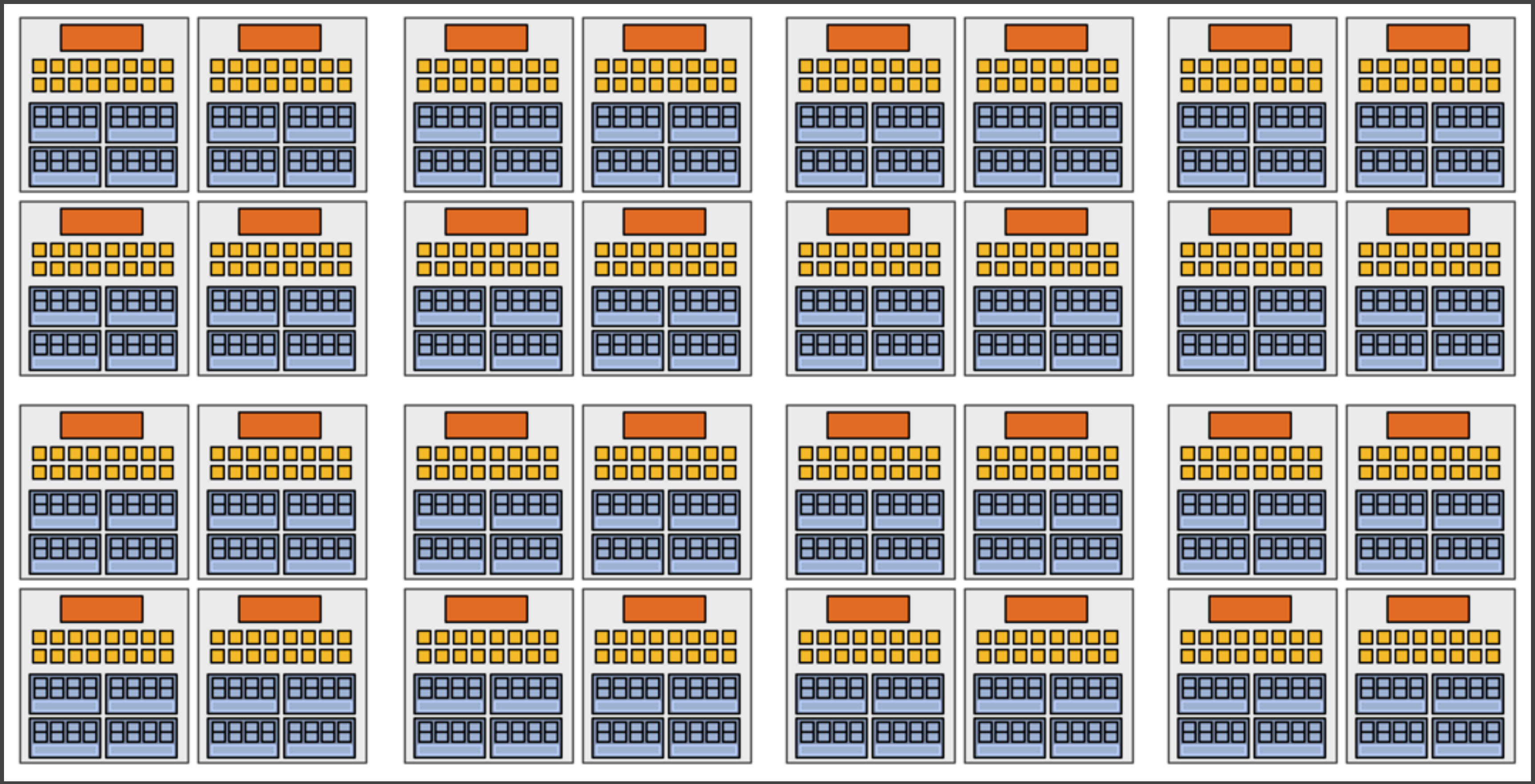


Graph courtesy of Bill Dally

GPUs Use Many Forms of Parallelism



“Extreme” Graphics Chip



16 cores x 32 SIMD functional units x 2 flops/cycle x 1 GHz = 1 TFLOP

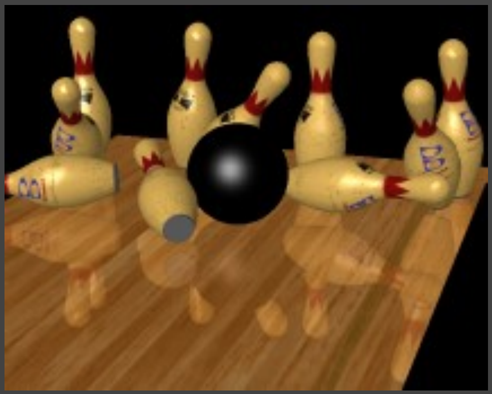
Application-Hardware Co-Design

Texture mapping must run at 100% efficiency

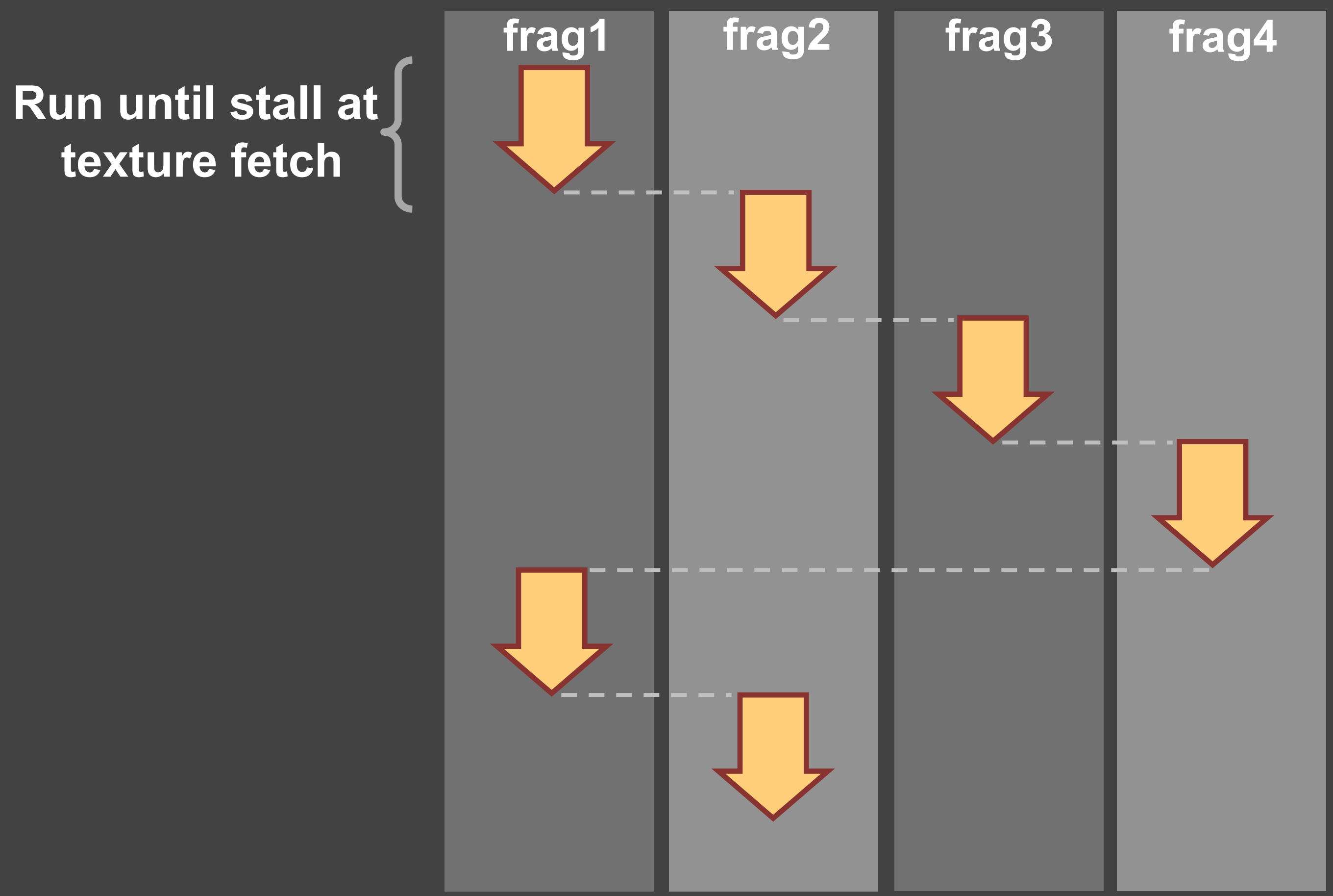
```
DCL       t0.xy          # Interpolate t0.xy
DCL       v0.xyzw        # Interpolate v0.xyzw
DCL_2D    s0             # Declaration – no code
TEX1D     r0, t0, s0     # TEXTURE LOAD!
MUL       r1, r0, v0     # Multiply
MOV       oC0, r1        # Store to framebuffer
```

Challenging

- Short inner loop (lots of branches)
- Random memory access (texture map)
- Very little temporal locality



GPU Multi-threading: Hide Latency



Fermi: 48 threads x 16 cores x 32 SIMD ALUs = 24,576 tasks

NVIDIA Historicals

Year	Product	Tri rate	CAGR	Tex rate	CAGR
1998	Riva ZX	3m	-	100m	-
1999	Riva TNT2	9m	3.0	350m	3.5
2000	GeForce2 GTS	25m	2.8	664m	1.9
2001	GeForce3	30m	1.2	800m	1.2
2002	GeForce Ti 4600	60m	2.0	1200m	1.5
2003	GeForce FX	167m	2.8	2000m	1.7
2004	GeForce 6800 Ultra	170m	1.0	6800m	2.7
2005	GeForce 7800 GTX	940m	3.9	10300m	2.0
2006	GeForce 7900 GTX	1400m	1.5	15600m	1.4
2007	GeForce 8800 GTX	1800m	1.3	36800m	2.3
2008	GeForce GTX 280			48160m	1.3
2010	GeForce GTX 480			42000m	0.9
2011	GeForce GTX 580			49400m	1.2
			1.7		1.7

SGI Historicals

Performance of Z-buffered rendering

Year	Product	Fragment	Rate	Triangle	Rate
1984	Iris 2000	100K	-	0.8K	-
1988	GTX	40M	4.5	135K	3.6
1992	RE	380M	1.8	2M	2.0
1996	IR	1000M	1.3	12M	1.6
			2.2		2.2

GPUs 10x More Efficient

# CPU cores	2 out of order	10 in-order
Instructions per issue	4 per clock	2 per clock
VPU lanes per core	4-wide SSE	16-wide
L2 cache size	4 MB	4 MB
Single-stream	4 per clock	2 per clock
Vector throughput	**8 per clock**	**160 per clock**

20 times greater throughput for same area and power

½ the sequential performance

Larrabee: A many-core x86 architecture for visual computing, D. Carmean, E. Sprangle, T. Forsythe, M. Abrash, L. Seiler, A. Lake, P. Dubey, S. Junkins, J. Sugerman, P. Hanrahan, SIGGRAPH 2008 (IEEE Micro 2009, Top Pick)

Software

Software is Inefficient

A C program – base line

A ruby/php program – 100x slower

A well-written C program – 10x faster

A crazy assembly language program – 2x-5x faster yet

Big Challenge

Graphics hardware specialization(s)

Multiple implementations with different characteristics

Software needs to be optimized for each platform

The resulting software is

- not portable
- costly to develop

Heterogeneous Platforms

LANL IBM Roadrunner

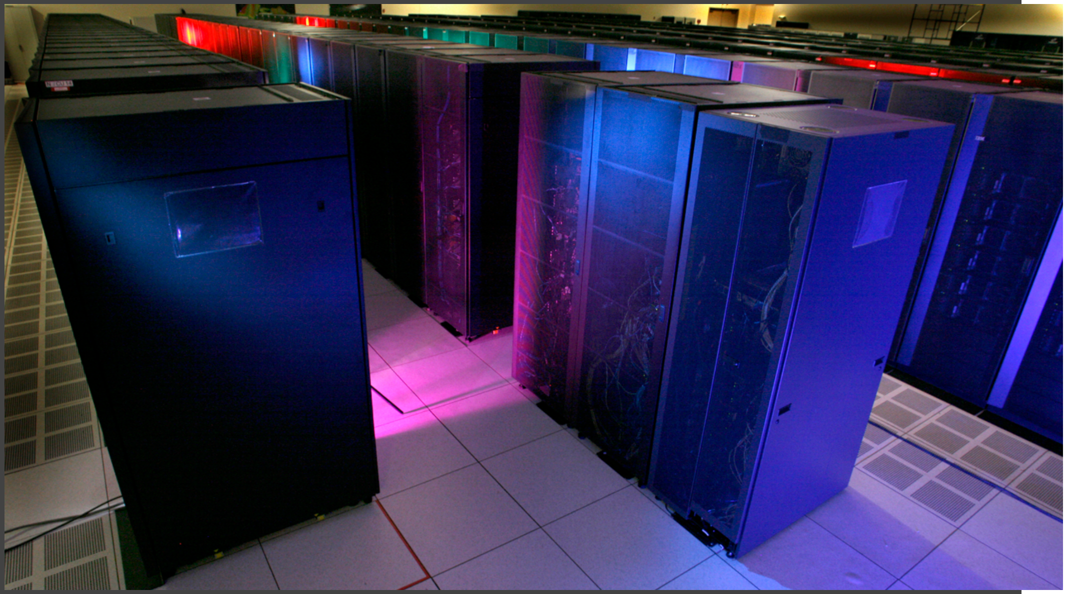
(Opteron + Cell)

Tianhe-1A

(Xeon + Tesla M2050 +

NUND 160GBps)

ORNL Titan



Even Bigger Challenges Ahead

Specialization leads to hybrid or heterogeneous systems

Heterogeneity leads to combinatorial complexity

Complexity makes it even harder to develop software

How Do We Handle Heterogeneity?

Program at a Higher-Level!

Graphics Libraries are High-Level

```
glPerspective(45.0);
for( … ) {
  glTranslate(1.0,2.0,3.0);
  glBegin(GL_TRIANGLES);
    glVertex(…);
    glVertex(…);
    …
  glEnd();
}
glSwapBuffers();
```

OpenGL "Grammar"

<Scene> = <BeginFrame> <Camera> <World> <EndFrame>

<Camera> = glMatrixMode(GL_PROJECTION) <View>

<View> = glPerspective | glOrtho

<World> = <Objects>*

<Object> = <Transforms>* <Geometry>

<Transforms> = glTranslatef | glRotatef | …

<Geometry> = glBegin <Vertices> glEnd

<Vertices> = [glColor] [glNormal] glVertex

Advantages

Portability

- Runs on wide range of GPUs

Advantages

Portability

Performance

Carefully designed to map efficiently to hardware

“Driver-Compiler” uses domain knowledge

- Vertices/Fragments are independent
- Textures are read-only; texture filtering hw
- Efficient framebuffer scatter-ops
- …

Advantages

Portability

- Allows hardware innovation

Performance

Advantages

Portability

Performance

Productivity

- Graphics libraries are easy to learn and use

Advantages

Portability

Performance

Productivity

Having your cake and eating it too!

Can We Apply this Idea to Scientific Computing?

Liszt

Z. DeVito, N. Joubert, M. Medina,

M. Barrientos, E. Elsen, S. Oakley,

J. Alonso, E. Darve, F. Ham, P. Hanrahan

"…the most technically advanced and perhaps greatest pianist of all time… made playing complex pieces on the piano seem effortless…"

Liszt: Solving PDEs on Meshes

```
val pos = new Field[Vertex,double3]
val A = new SparseMatrix[Vertex,Vertex]

for( c <- cells(mesh) ) {
    val center = avg(pos(c.vertices))
    for( f <- faces(c) ) {
        val face_dx = avg(pos(f.vertices)) – center
        for ( e <- f edgesCCW c ) {
            val v0 = e.tail
            val v1 = e.head
            val v0_dx = pos(v0) – center
            val v1_dx = pos(v1) – center
            val face_normal = v0_dx cross v1_dx
            // calculate flux for face …
            A(v0,v1) += …
            A(v1,v0) -= …
```

Challenges in Compiling GP Language

Compiler needs to reason about

- Parallelism
- Locality
- Synchronization

Fundamentally, analyzing dependencies is hard

1. Analyzing functions: A[i] = B[pow(2,i) / mod(i,4) + f(i)]
2. Analyzing pointers: A[i] = *ptrA

Liszt Enables Dependency Analysis

Mesh neighborhood accessed through built-in functions

- Pattern of access defines stencil
- Stencil shape is fixed and can be determined by static analysis

Fields accessed consistently during loops

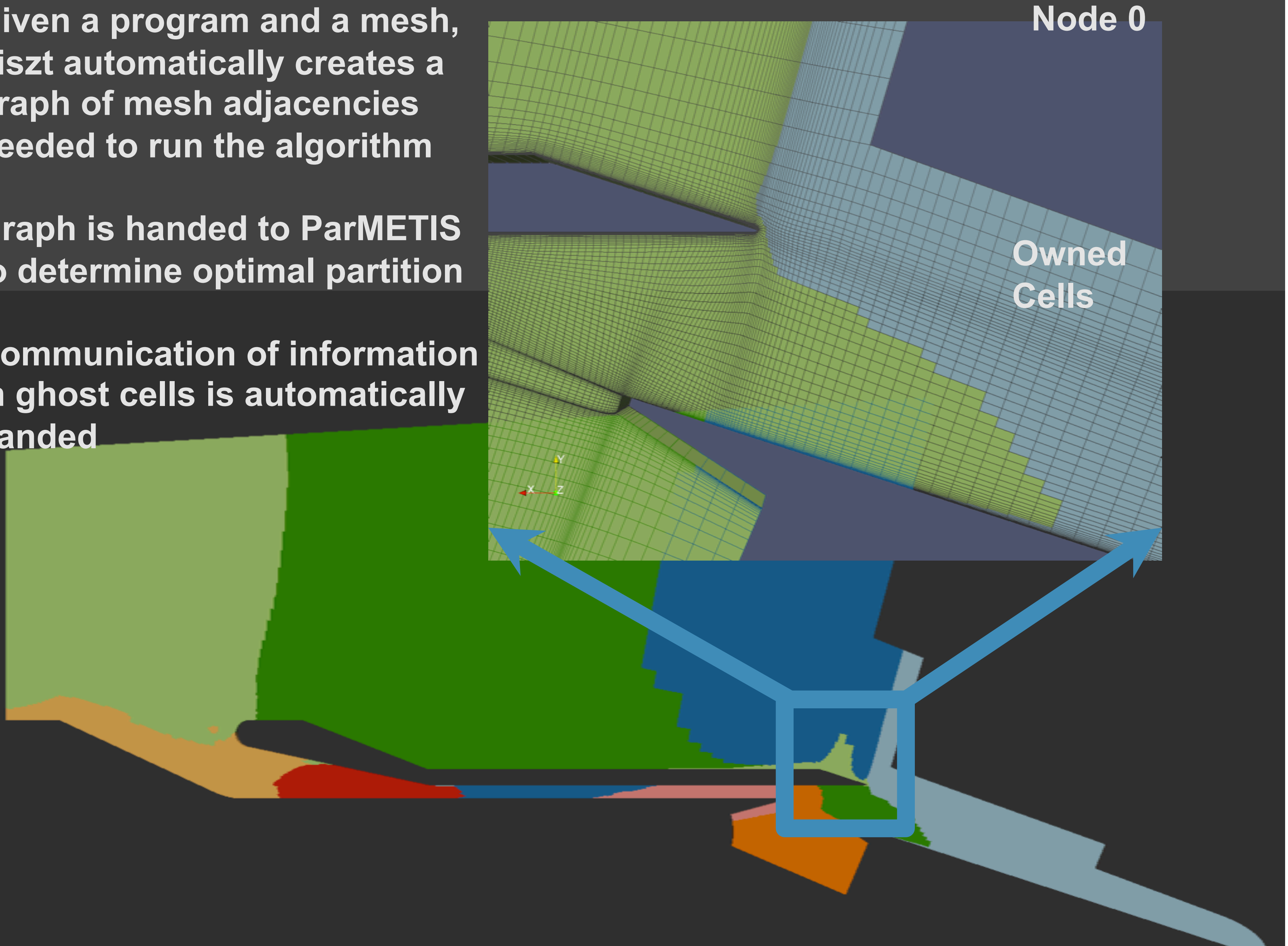
- Field accesses are organized into “phases”
- Within a forall, either read-only, write-only or reduce-only access pattern

Domain Decomposition / Ghost Cells

Given a program and a mesh, Liszt automatically creates a graph of mesh adjacencies needed to run the algorithm

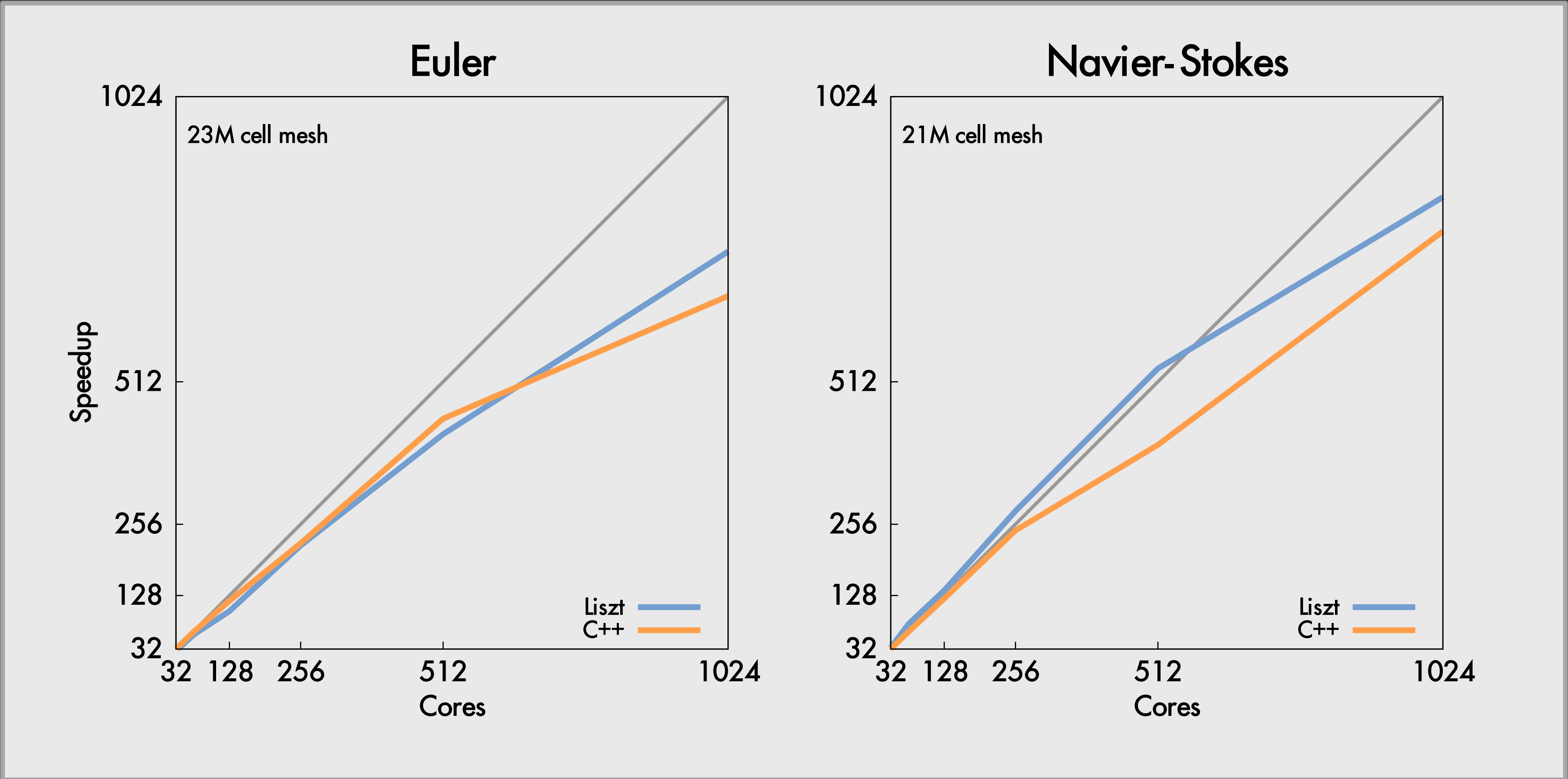
Graph is handed to ParMETIS to determine optimal partition

Communication of information in ghost cells is automatically handed



Scalable to Large Clusters

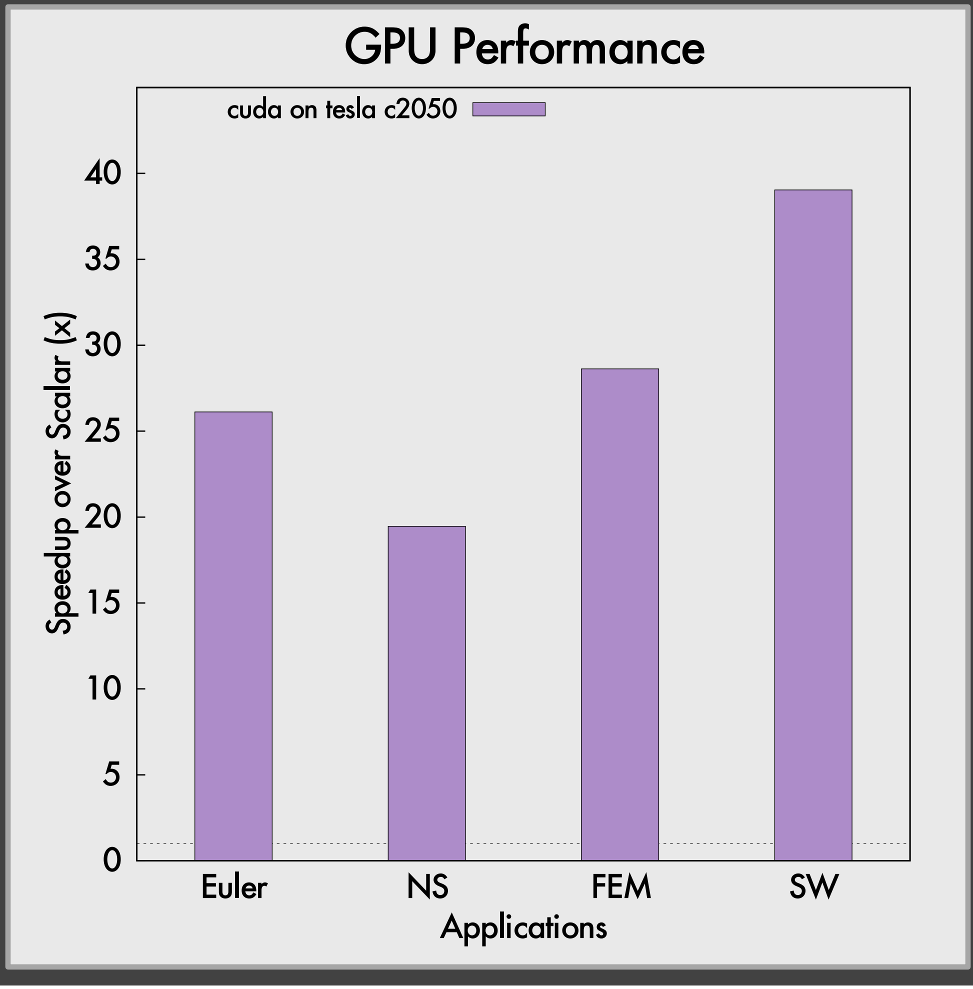
4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 16GB RAM per node. 256 nodes, 8 cores per node



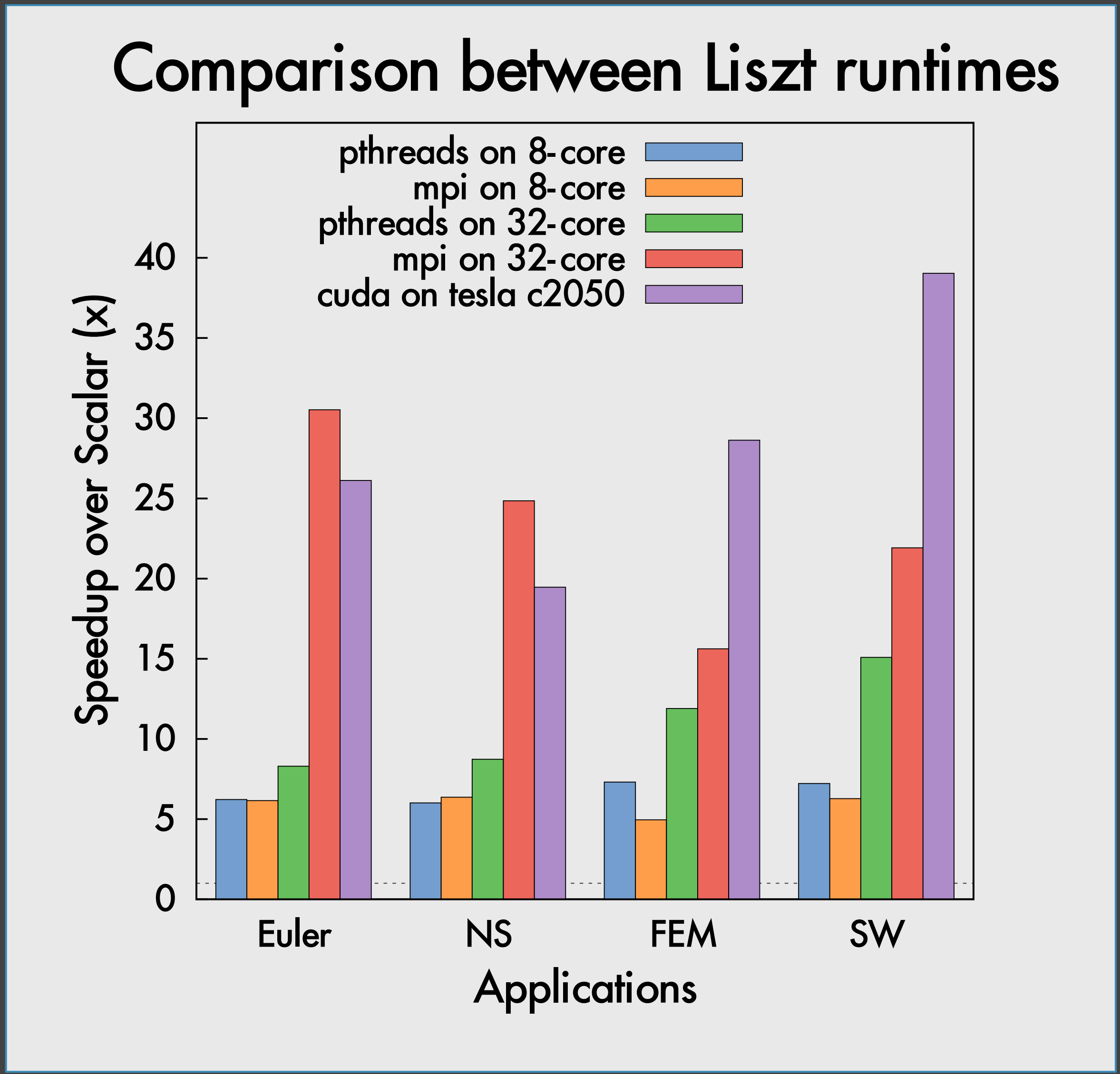
Runs Very Fast on GPUs

Tesla C2050 vs. 1 core Nehalem E5520 (2.26 Ghz)

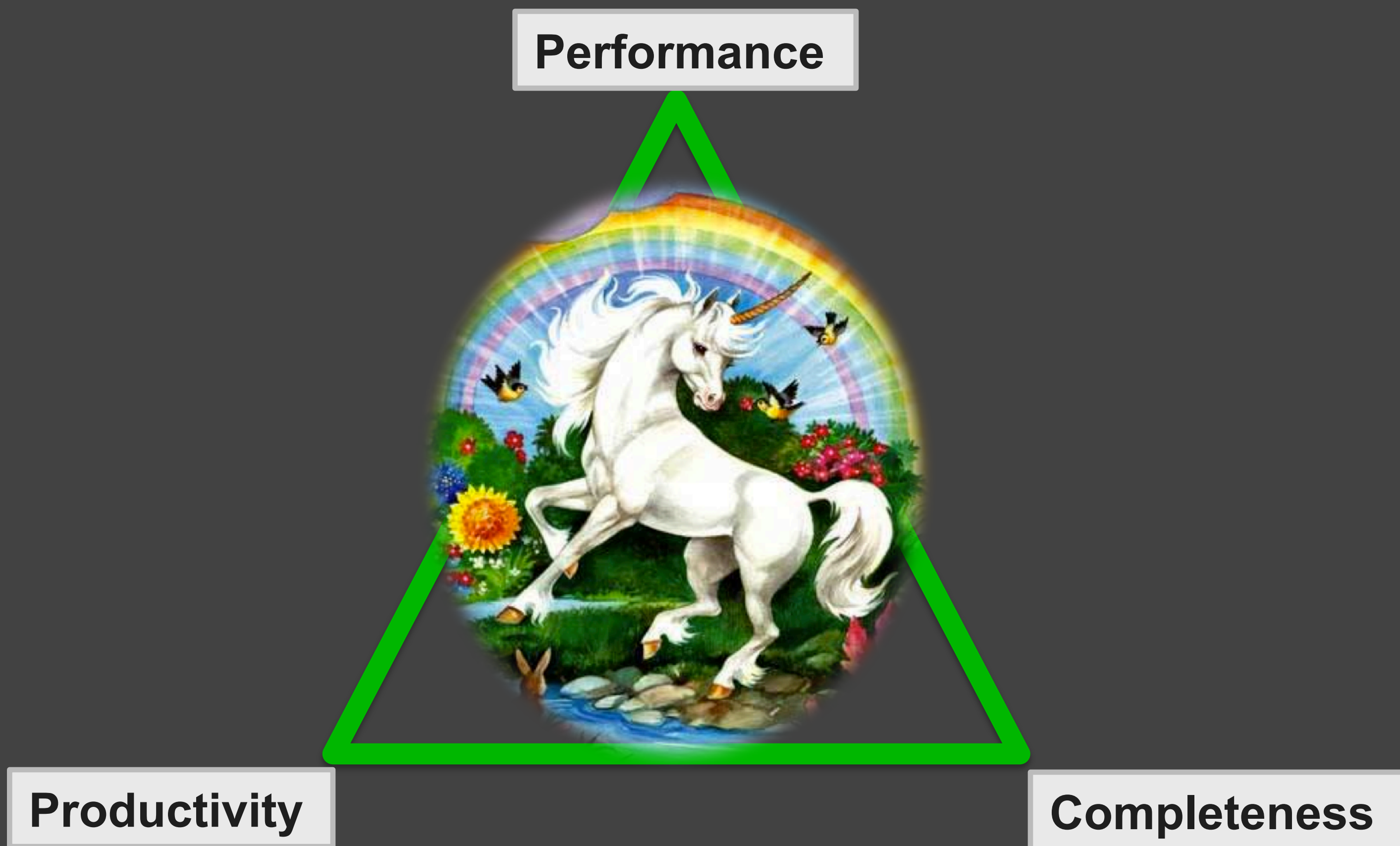
Double Precision



And Even SMPs

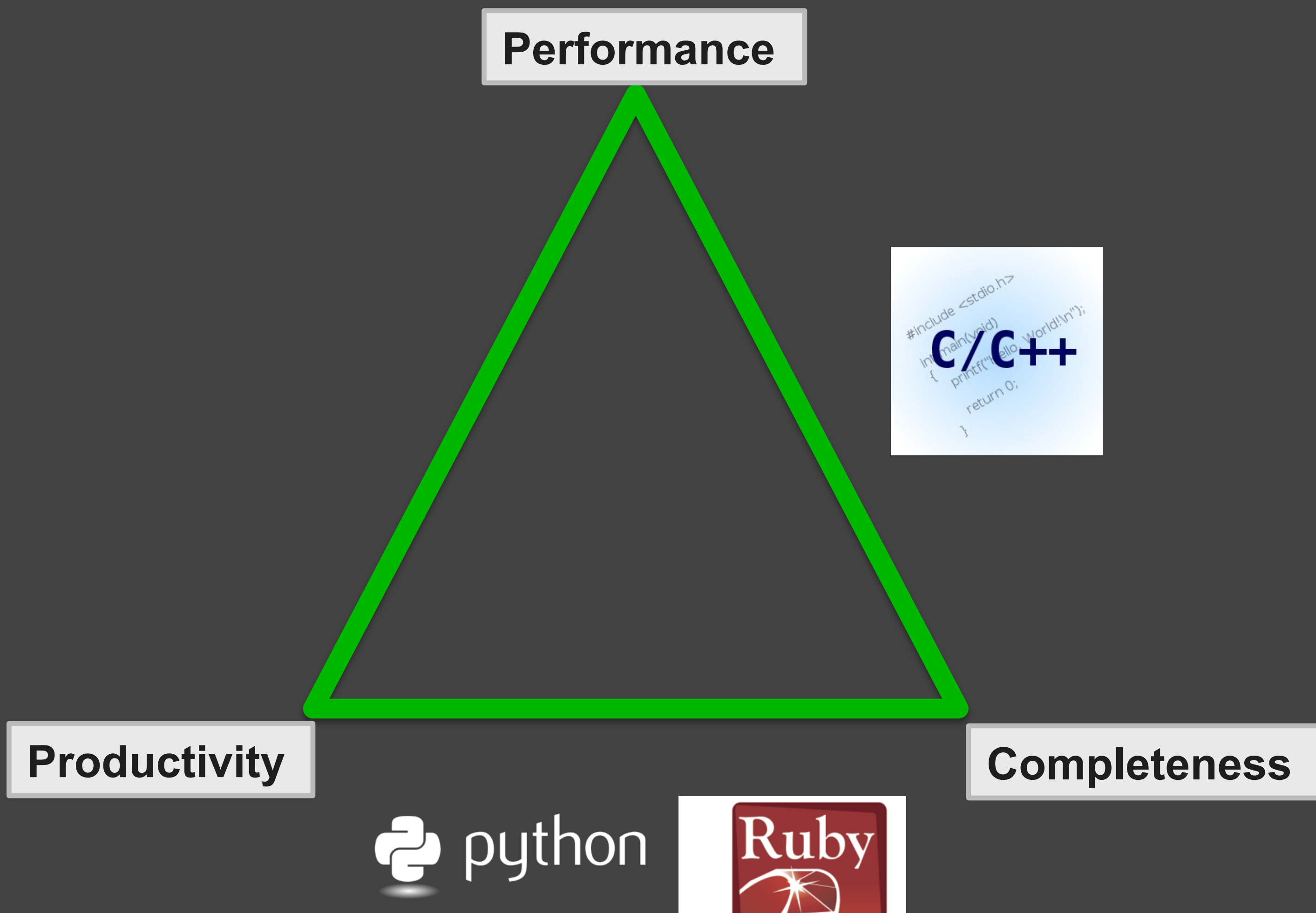


The Ideal Parallel Programming Language

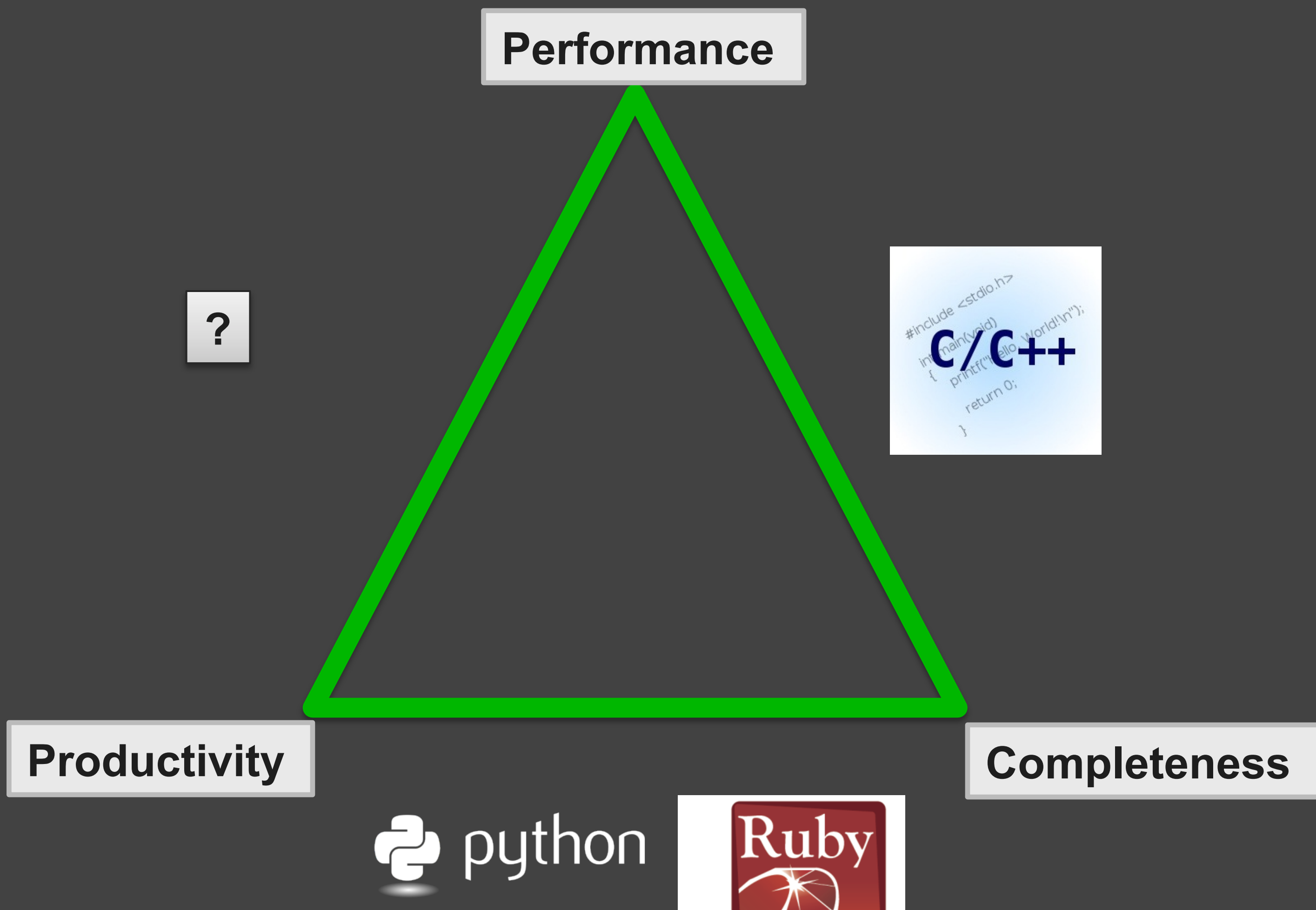


From Workshop on Concurrency for Application Programmers

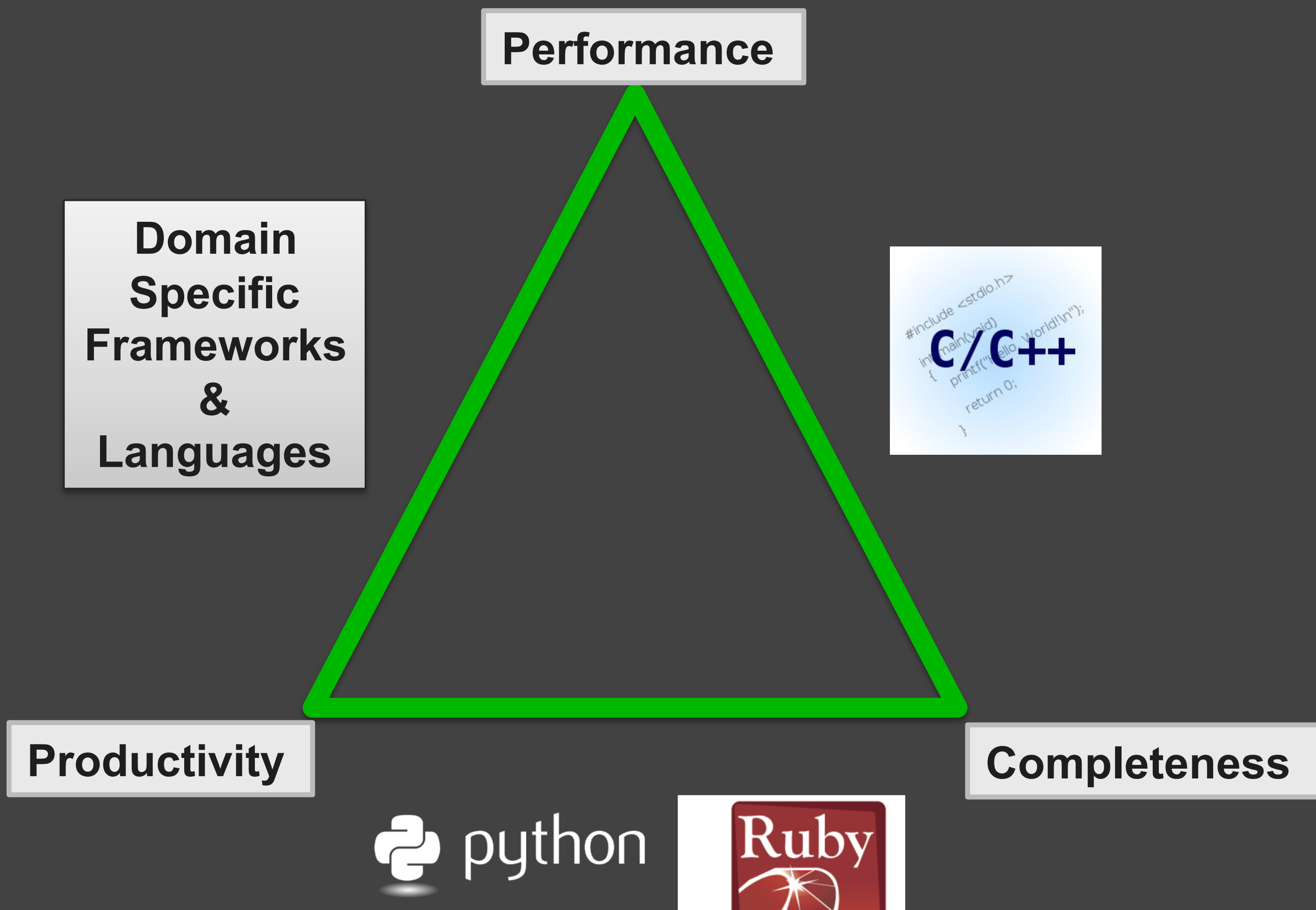
Successful Languages



Successful Languages



Additional Possibility



Wrap Up

Summary

Graphics systems require advanced simulation

Not having enough cycles forced us to be efficient

Both performance and portability are important

Leads to rapid evolution of innovative hardware

High-Level Abstractions

Applications are written using

- High-level frameworks: game engines
- Domain-specific languages: shading languages

Advantage of high level approach

- … makes programmers productive
- … allows efficient automatic parallelization

Careful Co-Design

This strategy works because of careful co-design of

- Applications (features)
- Algorithms
- Software
- Hardware

Thank you